

Build Tester – Version 1.4.7.0

Purpose:

Build Tester was created to simplify the speed testing of different Stockfish builds; however, it will work with any engine that outputs both nodes/sec and total nodes from their bench command. Its simple design is just a quick way of scheduling and comparing the results of different binaries when you must contend with non-deterministic fluctuations per run.

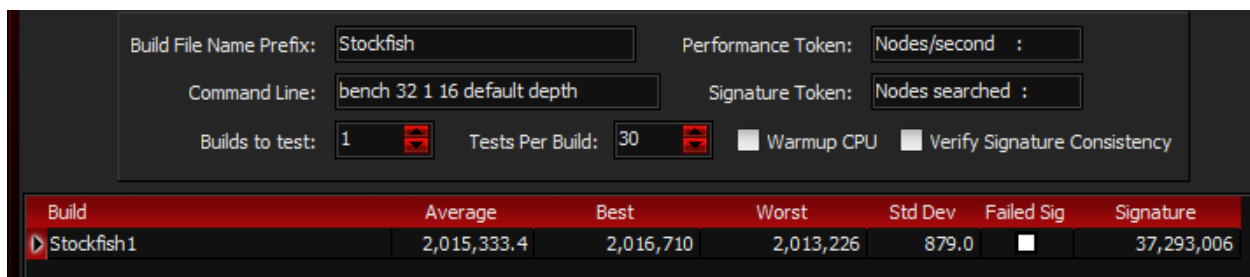
Environment Preparation:

It is important that you disable Turbo-Boost or equivalent technologies before running Build Tester or take other measures to stabilize your CPU clock speed. If you cannot disable Turbo-Boost, cooling the CPU to prevent it from reaching thermal max is also effective. For example, a laptop without Turbo-Boost settings in the BIOS can be placed on top of an air-conditioning vent. I have found this to lock the CPU's clock rate as long as the CPU remains at least 15 degrees below thermal max. Water cooled systems may also be stable with Turbo-Boost enabled. Using [Real Temp](#) to monitor clock rate and temperature during testing is important. (I am not aware of a similar application for AMD chips.)

In addition, if you are unable to lock your CPU clock, Build Tester provides a warmup CPU option. This runs your builds until more than 60 seconds have elapsed to get the CPU to a stable operating temperature. If you cannot disable Turbo-Boost, it is **critically important** to use the *Warmup CPU* option or the first engine test is guaranteed to add erroneous data.

Disabling hyper threading in the BIOS reduces variability for single threaded tests by as much as 61%; however, in my testing it made no difference when testing at the maximum physical core count. Set your windows power plan to maximum performance during testing.

Finally you should close and disable as many programs as possible so that the system is solely focused on running the tests. Booting into safe-mode is **highly recommended**. Manually killing tasks or stopping services can all reduce the workload on the CPU. In fact, in the example screen shot used in the User Interface section, the Standard Deviation was twice as high before booting into safe-mode. Disable your screensaver/blanker/lock timeout and do not use your machine during testing. Avoid touching your mouse or keyboard.



The screenshot shows the Build Tester application interface. At the top, there are configuration fields: 'Build File Name Prefix' set to 'Stockfish', 'Performance Token' set to 'Nodes/second', 'Command Line' set to 'bench 32 1 16 default depth', and 'Signature Token' set to 'Nodes searched'. Below these are 'Builds to test' set to 1 and 'Tests Per Build' set to 30. There are also checkboxes for 'Warmup CPU' and 'Verify Signature Consistency', both of which are unchecked. At the bottom, there is a table with the following data:

Build	Average	Best	Worst	Std Dev	Failed Sig	Signature
Stockfish1	2,015,333.4	2,016,710	2,013,226	879.0	<input type="checkbox"/>	37,293,006

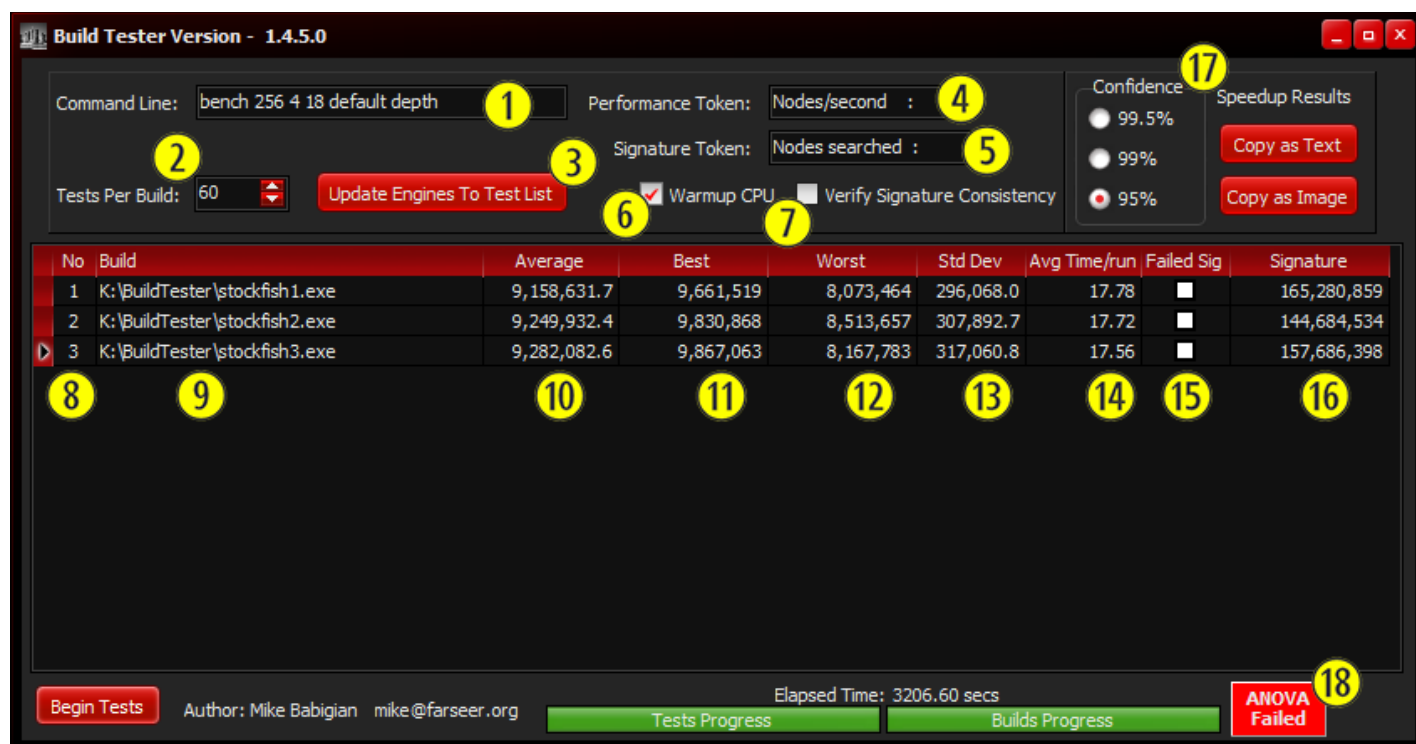
In the single threaded (deterministic) test above, you can see how low the standard deviation is once the machine has been stabilized. Only 0.17% difference between the best and worst run out of 30.

Verify your test environment after you've taken the suggested measures above by running a bench test lasting more than 15 seconds and 30 test runs. Divide the standard deviation by the average NPS. If the

result is less than 0.1%¹, your environment is in good condition. (Some CPU models actually show substantially lower variance than this.)

Finally, before beginning any test runs, you should open Task Manager and verify your CPU load is near or at zero. Users of this program have closed the program in the middle of a test run and left a full tilt Stockfish instance running in the background. Special code has been added to version 1.4.0.0 to warn the user if they attempt to close the application during testing and Build Tester will then gracefully shutdown after the current bench command underway completes.

User Interface:²



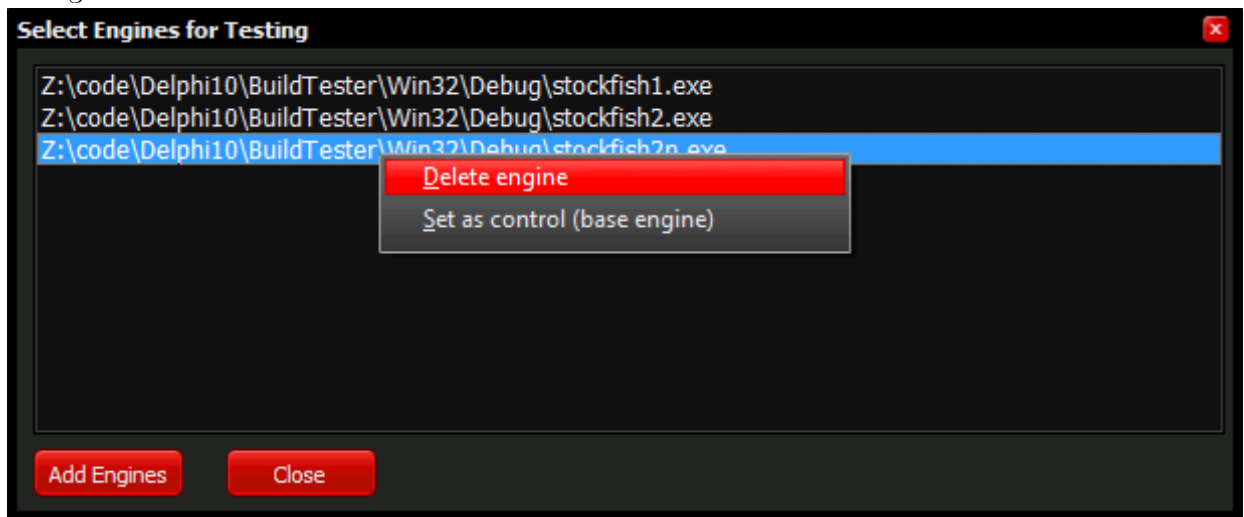
In order to use the tester, you'll need to create at least one EXE to test and place it in the same directory as the Build Tester. The UI features are described below.

1. **Command line** that will be passed to the engine when it is started. You can pass any valid command arguments in this box. In this example, I'm running the bench command with a hash size of 64, 6 threads, and searching to a depth of 18 ply. You should set the command line options such that **each test run takes at least 15 seconds to complete**. Shorter times will radically increase the standard deviation (reducing accuracy) and longer times will provide improved accuracy but diminishing returns. **DO NOT** use "default time."

¹ Updated from 0.04% after testing several more CPU models with higher variability.

² In the graphic above Stockfish2 and 3 are the exact same exe. Both are 1.55% faster than Stockfish1, but as you can see, 60 test runs were not enough to pass at the 95% confidence level due to the high variability from a 4 thread test. However, 100 test runs passed even at 99.5% confidence.

2. **Tests per build** is exactly that. How many times do you want to run each EXE. Assuming you've stabilized your PC to the minimum level recommended in the "Environment Preparation" section, and you are testing using 4 threads, **more than 60 tests** will be required to "reliably" determine with 95% confidence a speed difference as small as 1.5%. If you are trying to measure less than a 1.5% difference, more than 100 tests may be required for multi-threaded tests. Single threaded measurements contain 30 to 50 times less variability and can provide reasonable resolution with as few as 10 tests.
3. **Update Engines To Test List** - This button allows you to add engines to test. Pressing it will bring up the dialog box below:



You can add engines with the Add Engines button, or select an engine as I did above (blue) and right-click to bring up the local menu. You can either delete the engine from the list or set it as the control for your experiments. Setting it as the control/base engine simply moves it to the top spot. All engines will be compared against the engine occupying the top spot. You can multi-select (Ctrl or Shift Click) in both this dialog and in the Open dialog. Multi-selecting engines is only useful when adding or deleting. If you multi-select and then use "Set as control," the last engine you clicked will be set as the control. No more than 11 engines are permitted. If you multi-select more than 11 engines, the first 11 will be added.

4. **Performance token** is the text string the tester is looking for in the engine's output. It assumes the value it is looking for begins after this token. If you are testing a different program that uses a different string, you can replace this one. This string must match the engine's output exactly.
5. **Signature token** is the text that precedes the total nodes searched. This figure is used for comparing builds when using deterministic command lines (single threaded).
6. **Warmup CPU** - If you are unable to lock your CPU clock, you can use this checkbox to run all the builds at least once in order to reach a stable operating temperature. If you are testing fewer than 6 EXE's, it will loop through all builds between 2 and 4 times which will run the engine(s) between 8 and 10 times before starting the official test run. However, in order to prevent excessively long warmup times, I now check after each bench run to see if we have exceeded 60 seconds. If more than 60 seconds have elapsed, the warmup phase ends and testing begins
7. **Verify signature consistency** is only useful when running deterministic command line options. It simply tells the program to verify that all test runs of the same EXE, generate the same signature. The only use for this is tracking down an intermittent bug (perhaps created by overly aggressive optimizations).

8. **Build No** – Used to relate the build to the results generated by the “Copy as” buttons. In order to keep the results to a reasonable width, this number is used for results comparison.
9. **Build** - Name of build EXE.
10. **Average** nodes per second of all runs.
11. **Best** nodes per second speed of all runs.
12. **Worst** speed of all runs.
13. **Std Dev** – Standard deviation of all runs.
14. **Average time/Run** – average time each bench command took to run. This can be used to show average Time to Depth if you use a depth based command line. If you use “Bench 64 4 1000 default time” you are getting average elapse time per run (Time is not recommended). Changing to “Bench 64 4 18 default depth” is still average time per run, but this also happens to be the average Time to Depth 18.
15. **Failed sig** – this column is only used if “Verify signature consistency” is checked. If all test runs for that EXE do not **have** the same signature, the box in this column will be checked.
16. **Signature** – The signature from the final test run. If you build 5 EXEs with different levels of optimization, you can run 1 test run each and a deterministic command line (i.e. “bench”) and visually see if all the signatures for each build match.
17. **Speedup Results** – These buttons allow you to copy the speedup results to the clipboard. You can select either 99.5, 99 or 95% confidence levels to be applied to the resulting data. 99% is the default. For these buttons to be enabled, you must test at least 2 builds with a minimum of 3 tests per run. The Copy as Image button will not be available if you test more than 9 builds. See the graphic below of what is placed in the clipboard and written to Results.jpg:

```
Build Tester: 1.4.6.0
Windows 7 Service Pack 1 (Version 6.1, Build 7601, 64-bit Edition)
Intel(R) Core(TM) i7-3970X CPU @ 3.50GHz
SafeMode: No
Running In VM: No
HyperThreading Enabled: Yes
CPU Warmup: Yes
Command Line: bench 512 1 16 default depth
Tests per Build: 10
ANOVA: Passed
```

	Engine# (NPS)	Speedup	Sp	Conf. 99.5%	S.S.		
1	(2,101,850.3)	---> 2	(2,059,982.3)	---> 2.032%	2,076.8	Yes	No
1	(2,101,850.3)	---> 3	(2,078,677.8)	---> 1.115%	1,831.6	Yes	No

The Build Tester Version, OS version, processor type (as identified by the OS) and whether the tests were run in Safe Mode are included. I also attempt to identify if the tests were run inside a virtual machine (a big no-no). I tested this VM detection mechanism using VMWare, however the program should be able to also identify, Wine, VirtualBox, and VirtualPC. (If anyone tests these, please let me know the results of the VM detection logic). I also attempt to determine if Hyper-threading is enabled. This worked on my PCs; however, many architectures could potentially give erroneous results. (i.e. I don't have a NUMA machine handy at the moment for testing.) Hyper-Threading detection definitely fails inside a virtual machine, but you should not be running Build Tester in a VM anyway. You can also see that the program documents whether a CPU warmup was used, the bench command given, how many tests per build were requested, and the ANOVA result (n/a, Passed, Failed).

The results include the engine numbers (the faster engine is always displayed on the left), the calculated

speedup, and Sp (pooled estimate of the common population standard deviation). All engines are compared to engine 1 only. Sp can be used to see how stable the test environment was – especially for single threaded tests. I’m documenting the following formulas so that when I look back at this program a year from now, I remember what I did. Sp is calculated with the following formula:

$$Sp = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}}$$

Dividing Sp by the average speed of the two engines should result in figures below 0.1%. (The example above was not run in Safemode, hence the high Sp figure). Conf xx% is a boolean result that is true (yes) if the difference between the engines meets or exceeds the confidence level you selected.

The confidence is a two tailed t-test. The value of t is calculated with the following formula:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

X₁ and X₂ are the two engine’s mean NPS and S₁ and S₂ are their standard deviations. n₁ = n₂ and n = the number of test runs. Degrees of freedom are calculated with the following:

$$d.o.f. = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\left(\frac{s_1^4}{n_1^2(n_1 - 1)} + \frac{s_2^4}{n_2^2(n_2 - 1)}\right)}$$

The value t is then compared to critical t values based on your selected confidence level and the degrees of freedom. If the calculated t is greater than the critical t value, your results are significant.

The final column is marked S.S. (Statistically Significant). This indicates whether the results are considered by “Build Tester” to be statistically significant using the following criteria. 1) Conf xx% = yes (passed t test), 2) If three or more builds were tested, ANOVA must have passed. 3) The tests were run in Safe Mode, 4) The tests were not run inside a virtual machine. 5) The time per run for the fastest running build is greater than 15 seconds, and at least 60³ test runs were performed. StockFish currently uses 37 test positions during the bench test. This means that each position gets but a tiny fraction of the total bench run time. Extremely fast bench tests have higher random variability.

³ 60 tests are generally unnecessary for single threaded tests. 10 or more should be fine for single-threaded tests unless the speed difference being measured is very small. As long as the lower bound confidence number is greater than zero and (if testing 3 or more builds) ANOVA passed, the results can be relied upon. The 60 test minimum is enforced to prevent **multi-threaded** non-sense speed-up claims. A multi-threaded 60 test run of two builds will take about 32 minutes. A small price to pay to keep the Internet free from misinformation.

18. This ANOVA block is gray at program start or if ANOVA (analysis of variance) is not used. If ANOVA is used (three or more builds and at least three tests each), this block will either turn green and say “passed” or red and say “failed.” ANOVA passes if at least one build tested varies significantly in speed from one of the others. Significance is determined by the confidence radio block. After running tests, selecting different confidence levels will update and potentially change the result shown in this block.

Program settings to reduce/overcome noise during testing

While using and writing this tool, I ran numerous tests and have determined some guidelines regarding run time per build and number of runs for reducing noise and improving results.

For testing large differences between builds or where time is short:

- Always use depth not time (i.e. Bench 32 1 18 default **depth**). The time method creates way more variability per run.
- Set bench depth to generate greater than 15 seconds per run (less than 20 ply for multi-thread tests if possible)
- Minimum 60 runs

For testing small differences or when accuracy is paramount:

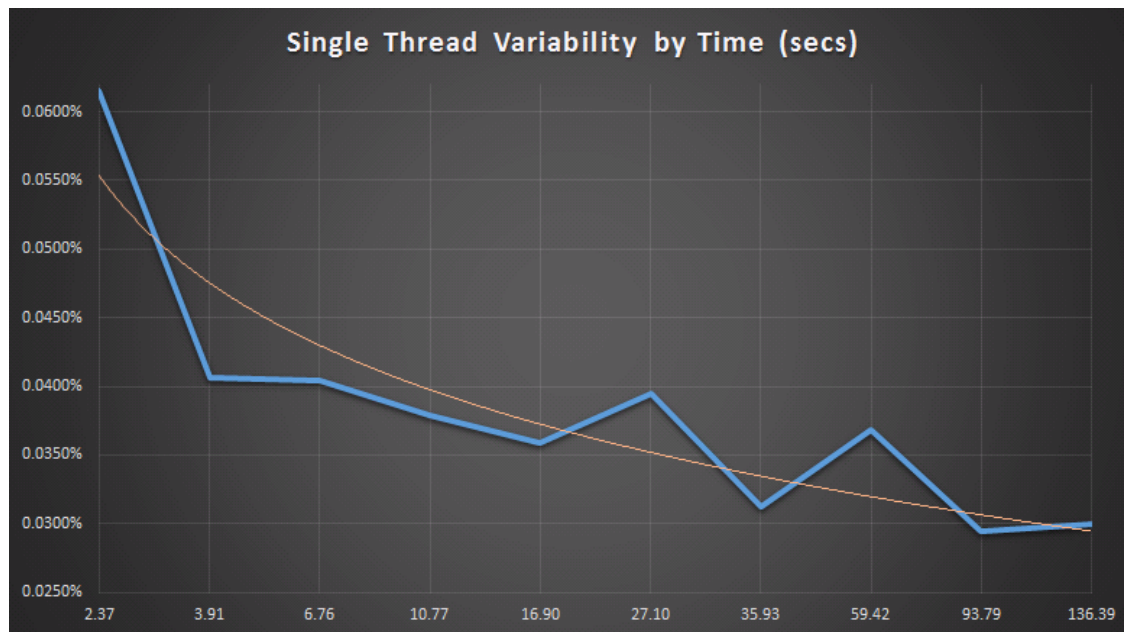
- Always use depth not time (i.e. Bench 32 1 18 default depth)
- Greater than 90 seconds per run (difficult to make a recommendation for multi-threaded tests as these run times will likely put you higher than 20 ply).
- Minimum 100 runs (more for multi-threaded tests, especially if depth > 19)

The 15 second rule has already been mentioned several times. To support this figure note the single thread test series below⁴:

Depth	Avg Time/run (secs)	Average NPS	Std Dev	Std Dev %
12	2.37	2,059,511.0	1,268.2	0.0616%
13	3.91	2,047,211.0	832.4	0.0407%
14	6.76	2,060,507.6	833.6	0.0405%
15	10.77	1,983,093.9	752.0	0.0379%
16	16.90	1,956,071.2	701.4	0.0359%
17	27.10	2,040,118.2	805.8	0.0395%
18	35.93	1,985,773.7	621.3	0.0313%
19	59.42	1,971,039.4	726.9	0.0369%
20	93.79	1,973,076.4	581.6	0.0295%
21	136.39	1,908,841.1	571.7	0.0300%

As you can see, the variability from a 2 second run (at depth 12) compared to times greater than 15 seconds show the amount of variability is roughly halved. The chart below is also worth a look.

⁴ Both the single and four thread tables were generated averaging only 30 test runs on an old i7 975.

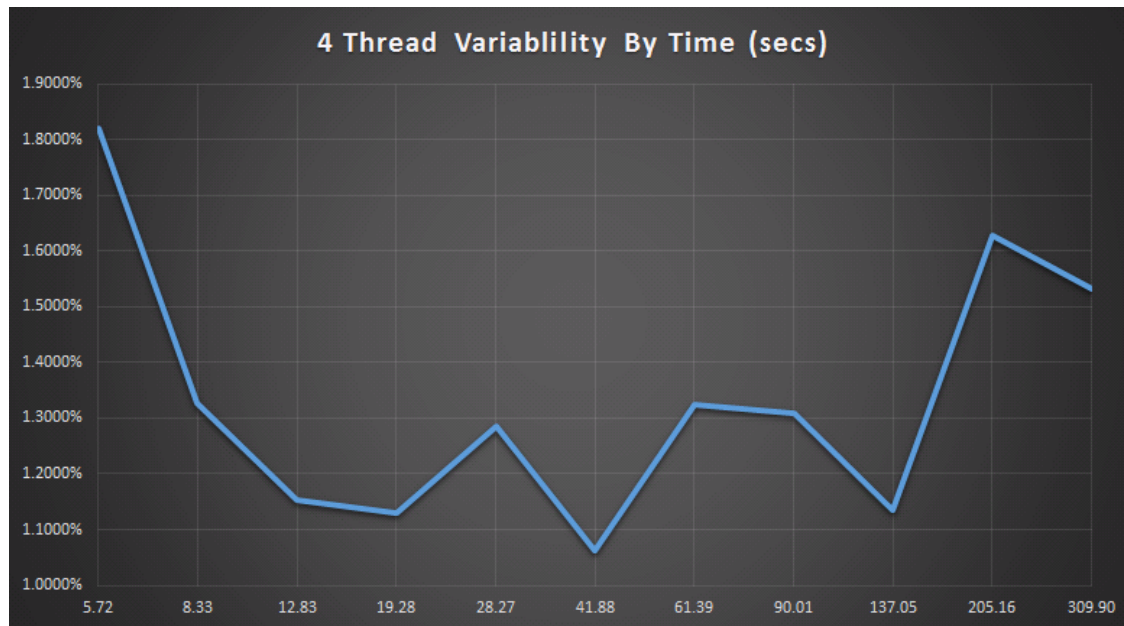


One interesting thing shown is that once you get past the noisy sub 15 second runs, the odd numbered plies appear to show more variability than you would expect compared to the even numbered ones. I was unable to explain this phenomenon but perhaps it is worth further investigation by the developers **if others can confirm this result**. The orange line is simply a logarithmic trend line. Single threaded test runs exceeding 90 seconds showed no significant improvement in reducing variability.

The next set of tests were conducted using **four threads**.

Depth	Avg Time/run (secs)	Average NPS	Std Dev	Std Dev %
16	5.72	6,597,237.0	119,997.6	1.8189%
17	8.33	6,682,166.2	88,731.7	1.3279%
18	12.83	6,732,277.6	77,604.6	1.1527%
19	19.28	6,835,297.7	77,177.6	1.1291%
20	28.27	6,906,910.2	88,737.6	1.2848%
21	41.88	6,972,475.7	74,099.2	1.0627%
22	61.39	6,971,859.9	92,359.3	1.3247%
23	90.01	6,979,948.1	91,394.0	1.3094%
24	137.05	6,934,302.6	78,798.7	1.1364%
25	205.16	6,932,641.7	112,936.4	1.6291%
26	309.90	6,902,384.3	105,836.1	1.5333%

First, notice that the increased variability caused by SMP's non-determinism is now more than **30 times higher** than the single threaded tests. Looking at the chart below we see the same improvement as time increases until we reach depths in the 20s (28.27 seconds on the chart).



At higher depths the variability starts to increase making measuring differences in builds problematic. I haven't a clue why this happens. Some wild theories would include, changes in the split point behavior as depth increases, etc. Most SF testing is done at game 60 or faster. Dividing the average time per run of the tests at 22 ply or deeper by 37 test positions indicates that the framework normally doesn't test patches at these depths. So perhaps this behavior is going unnoticed. I point it out only because I found it odd. It doesn't necessarily mean anything is wrong.

ANOVA and Statistical Landmines:

Although ANOVA calculations have been added to Build Tester, this does not fully address Lyudmil Antonov's issue with multiplicity. To reduce the problems involved with comparing multiple EXE's I have included this section, and made the following program restrictions.

- 1) You can now test no more than 11 builds at a time.
- 2) I only compare each build against the first build. This reduces the number of comparisons to (builds - 1) and reduces the likelihood of committing a type I error. If you are testing a patch, the first engine should be the master or base build. If you are testing different compilers or compiler/linker directives, the first engine should be your control. If testing a bunch of profile builds, each will be compared to the first engine only.

When testing three or more builds, the likelihood of you committing a type I error can be calculated with the following formula:

$$\alpha_{FW} = 1 - (1 - \alpha_{PC})^c$$

α_{pc} = per comparison error rate (0.05 if you selected 95% in the confidence block, 0.01 if you selected 99%, 0.005 if you selected 99.5%)

c = the number of comparisons. (4 builds would mean 3 comparisons).

Therefore, if you want to reduce the likelihood of committing a type I error to 5% or less among all the comparisons made, you should select 99% in the confidence radio block for between 3 and 6 builds (maximum of 5 comparisons). If you are testing between 7 and 11 builds, 99.5% should be chosen to avoid exceeding 5%. Unfortunately, this also increases the likelihood of a false negative result (not detecting a legitimately faster build in your test group).

To avoid these issues altogether, you should test no more than two builds at a time (paired tests).

Final Thoughts:

Due to the non-determinism inherent in chess engines, it takes many runs to get a reasonably stable measurements of relative speed. Despite the calculation of confidence levels, the results will be/are meaningless if the user runs them in an improperly stabilized environment. I proved this repeatedly during my own testing. I could easily get results that were 95 or 99% confident, testing the exact same EXE if the underlying machine environment was improper. This is why I added checks for safe mode, and virtual environments. Unfortunately, there is no possible way to check for every possible way in which the user can garbage up the results. Therefore I implore those posting results from their test runs to verify the stability of their environment is below the 0.1% threshold mentioned in the Environment Preparation section.

The program writes the table data to BuildScores.CSV. This makes it a bit easier if you want to massage the data in Excel.

After spending the time to create this tool for some internal testing, I thought someone else might find it helpful so I'm plopping it onto the Internet.

Have fun,
-Mike